HTTP Example

Copyright ©: University of Illinois CS 241 Staff

HTTP Request Framing

- Characteristics
  - ASCII-based (human readable)
  - Framed by text lines
  - First line is command
  - Remaining lines are additional data
  - Blank line ends request frame

```
GET     /surf/too/much.html     HTTP/1.0
Date: 28 February 2000 011:25:53 CST
Server: www.surfanon.org
<blank line>
```


Copyright ©: University of Illinois CS 241 Staff

HTTP Server Push (Netscape-Specific)

- Idea
  - Connection remains open
  - Server pushes down new data as needed
  - Termination
    - Any time by server
    - Stop loading (or reload) by client
- Components
  - Header indicating multiple parts
  - New part replaces old part
  - New part sent any time
  - Wrappers for each part

Copyright ©: University of Illinois CS 241 Staff

HTTP Server Push (Netscape-Specific)

```
HTTP/1.0   200   OK
Content-type: multipart/x-mixed-replace;\
boundary=---never_in_document---

---never_in_document---
```

the data component

```
Content-type: text/html

(actual data)
---never_in_document---
```


Copyright ©: University of Illinois CS 241 Staff

Example

- Push server
  - Client-server connection remains open
  - Server pushes new data
- Use pthreads
- Main thread
  - Accepts new client connections
  - Spawns child thread for each client
- Child threads
  - Parses client requests
  - Constructs response
  - Checks for file modification
  - Pushes file when necessary

Copyright ©: University of Illinois CS 241 Staff

Example: Server Thread Flow Chart

Start → Set up server TCP socket → Initialize thread attributes structure → Create a thread specific data structure → Wait for a connection → Spawn a child thread to handle new connection → (back to) Create a thread specific data structure

Example: Client Thread Flow Chart

Start

Wait for 1 second or until client send request

Got client request?

n → Need to send?

y → Request valid?

Request valid? y → Store file name and reset send time → Need to send?

Request valid? n → Done

Need to send? n → Wait for 1 second or until client send request

Need to send? y → Send file to client

Send file to client → Success in send?

Success in send? y → Wait for 1 second or until client send request

Success in send? n → Done

Copyright ©: University of Illinois CS 241 Staff

set_up_server_socket

```
static int set_up_server_socket (u_short port) {
    int fd;                  /* server socket file descriptor   */
    int yes = 1;             /* used for setting socket options */
    struct sockaddr_in addr;  /* server socket address          */

    /* Create a TCP socket. */
    if ((fd = socket (PF_INET, SOCK_STREAM, 0)) == -1) {
        perror ("set_up_server_socket/socket");
        return -1;
    }

    /* Allow port reuse with the bind below. */
    if (setsockopt (fd, SOL_SOCKET, SO_REUSEADDR,
                    (char*)&yes, sizeof (yes)) == -1) {
        perror ("set_up_server_socket/setsockopt");
        return -1;
    }
```


Copyright ©: University of Illinois CS 241 Staff

set_up_server_socket

```
    /* Set up the address. */
    bzero (&addr, sizeof (addr));
    addr.sin_family      = AF_INET;     /* Internet address          */
    addr.sin_addr.s_addr = INADDR_ANY;  /* fill in local IP address  */
    addr.sin_port        = htons (port); /* port specified by caller*/

    /* Bind the socket to the port. */
    if (bind (fd, (struct sockaddr*)&addr, sizeof (addr)) == -1) {
        perror ("set_up_server_socket/bind");
        return -1;
    }

    /* Listen for incoming connections (socket into passive state). */
    if (listen (fd, BACKLOG) == -1) {
        perror ("set_up_server_socket/listen");
        return -1;
    }

    /* The server socket is now ready. */
    return fd;
}
```


Copyright ©: University of Illinois CS 241 Staff

wait_for_connections

```
static void wait_for_connections (int fd){
    pthread_attr_t attr;  /* initial thread attributes               */
    thread_info_t* info;  /* thread-specific connection information */
    int len;              /* value-result argument to accept        */
    pthread_t thread_id;  /* child thread identifier                */

    /* Signal a bug for invalid descriptors. */
    ASSERT (fd > 0);

    /* Initialize the POSIX threads attribute structure. */
    if (pthread_attr_init (&attr) != 0) {
        fputs ("failed to initialize pthread attributes\n", stderr);
        return;
    }
    /* The main thread never joins with the children. */
    if (pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED)
    != 0) {
        fputs ("failed to set detached state attribute\n", stderr);
        return;
    }
```


Copyright ©: University of Illinois CS 241 Staff

wait_for_connections

```
/* Use an infinite loop to wait for connections.  For each
connection, create a structure with the thread-specific data, then
spawn a child thread and pass it the data.  The child is
responsible for deallocating the memory before it terminates. */
while (1) {

/* Create a thread information structure and initialize
   fields that can be filled in before a client contacts
   the server. */
if ((info = calloc (1, sizeof (*info))) == NULL) {
     perror ("wait_for_connections/calloc");
     return;
}
info->fname      = NULL;
info->last_sent = (time_t)0;
```


Copyright ©: University of Illinois CS 241 Staff

wait_for_connections

```
    /* Wait for a client to contact the server. */
    len = sizeof (info->addr);
    if ((info->fd = accept (fd, (struct sockaddr*)&info->addr,
                            &len)) == -1) {
        perror ("accept");
        return;
    }
    /* Create a thread to handle the client. */
    if (pthread_create (&thread_id, &attr,
                        (void* (*) (void*))client_thread, info) != 0) {
        fputs ("failed to create thread\n", stderr);

        /* The child does not exist, the main thread must clean up. */
        close (info->fd);
        free (info);
        return;
    }
  }
}
```


Copyright ©: University of Illinois CS 241 Staff

client_thread

```
void client_thread (thread_info_t* info) {
    /* Check argument. */
    ASSERT (info != NULL);

    /* Free the thread info block whenever the thread terminates.
    Note that pushing this cleanup function races with external
    termination.  If external termination wins, the memory is never
    released. */
    pthread_cleanup_push ((void (*)(void*))release_thread_info, info);

    /* Loop between waiting for a request and sending a new copy of
    the current file of interest. */
    while (read_client_request (info) == 0 &&
        send_file_to_client (info) == 0);

    /* Defer cancellations to avoid re-entering deallocation routine
        (release_thread_info) in the middle, then pop (and execute) the
        deallocation routine.*/
    pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, NULL);
    pthread_cleanup_pop (1);
}
```

 Copyright ©: University of Illinois CS 241 Staff

client_has_data

```
static int client_has_data (int fd) {
    fd_set read_set;
    struct timeval timeout;

    /* Check argument. */
    ASSERT (fd > 0);

    /* Set timeout for select. */
    timeout.tv_sec = CHECK_PERIOD;
    timeout.tv_usec = 0;

    /* Set read mask for select. */
    FD_ZERO (&read_set);
    FD_SET (fd, &read_set);

    /* Call select.  Possible return values are {-1, 0, 1}. */
    if (select (fd + 1, &read_set, NULL, NULL, &timeout) < 1) {
     /* We can't check errno in a thread--assume nothing bad has happened. */
     return 0;
    }

    /* Select returned 1 file descriptor ready for reading. */
    return 1;
}
```


Copyright ©: University of Illinois CS 241 Staff